System Isolation and Network Fast-Fail Capability in Solaris

Gabriel Montenegro (gabriel.montenegro@eng.sun.com)

Steve Drach (steve.drach@eng.sun.com)

Sun Microsystems, Inc.

Abstract

UNIX hosts configured for network operation typically hang, or freeze, when temporarily disconnected from the network. This failure is unacceptable to the user of a mobile host who purposely disconnects the host from the network in order to move it to another location not serviced by a network connection. This paper describes an approach that automatically enables a system to continue to function, in a diminished capacity, when disconnected from the network.

1.0 Introduction - Why is there a problem?

Modern computing environments typically consist of a core cluster of servers and many desktop workstations. Together they implement a client-server model of decentralized distributed processing that depends on a robust functional network for successful operation. This type of distributed processing avoids reliance on a particular host and, instead, places the emphasis on the network. In other words, "the network is the computer."

The problem is, if a network operation fails, the system is often rendered unusable. It is assumed that this is a temporary condition, and the best recovery procedure is to periodically retry the failed network operation until conditions improve. Some clients, such as NIS, ignore network failures and retry forever, effectively preventing their clients from continuing to operate, even in a diminished capacity. In most cases, the system appears to hang or freeze during the period of disconnection.

With the advent of nomadic computing devices, systems operating with intermittent network connectivity are increasingly more common. In this case, system isolation brought on by network disconnection may very well be purposeful and expected to continue for the duration of the user's session. Any network operations are guaranteed to fail. It is not, however, desirable for the system to stop functioning.

An ancillary problem is that several applications built specifically for the mobile or nomadic user have appeared. These typically have a disconnection state, in which they either log operations to be replayed upon reconnection, or make use of local caches. Each application determines system connectivity using a private method that typically depends on time-outs. This presents the user with different non-homogeneous models of system behavior during a network outage.

It has been pointed out that the main problem with distributed systems is that the applications are implemented with the assumption that all the processes involved reside in the same system. As soon as latencies, errors and network outages occur, the paradigm breaks down [WALDO 94], and errors such as system hangs appear,

Obviously, one approach to solving the problem is to modify existing services and applications so that they have more information regarding network status and so that they are more aware of the characteristics of the underlying communications medium. This approach, however, is very expensive and time consuming.

We chose, instead, to provide a fast-fail capability so that network operations that are guaranteed to fail, do so quickly and that the clients who are notified of the failures give up quickly. Instead of waiting for network operations to resume (and hanging during the outage), it makes more sense for the system to continue operating, using whatever local resources (e.g. cached information) may be available.

In addition to allowing currently existing applications an opportunity to continue operations during a network outage, we have also provided mechanisms for developing more knowledgeable applications and kernel modules that modify their behavior during a network disconnection.

The fast-fail capability can also alleviate the ancillary user interface problem described earlier, by providing these applications with immediate and unequivocal indication of network disconnection. They no longer need to implement private algorithms to determine the state of network connections.

2.0 What is System Isolation?

When thinking about system isolation, several related concepts and networking conditions come to mind:

- stand-alone systems
- time-outs
- temporary network outages
- Mobile IP
- network partitioning
- disabled interfaces
- weakly connected systems

We now examine each of these in some detail.

The problem we are trying to solve only concerns hosts that must adapt to intermittent network connectivity. This means that a system that is always disconnected from the network is not part of the problem domain. It is a stand-alone system that does not run the risk of freezing by retrying network operations.

System isolation is a guarantee that no network access is possible, and will not be possible for much longer than usual network time-out periods, typically ranging from under a second to several seconds. Accordingly, system isolation occurs when a time-out is on the order of minutes or hours.

Admittedly, the distinction between system isolation and a temporary network outage is sometimes hazy. The determination on whether a system is isolated or not can also depend on the particular network technology. For example, wireless interfaces can experience loss of signal in between cells, because of fading, etc. These are momentary and do not represent system isolation. Mobile IP [PERKINS 95] deals with this scenario to some extent by suggesting procedures for hand-off and for reducing the adverse effects of this unreliable link.

On the other hand, mobile systems may choose to operate in isolation for extended periods of time (e.g. during travel in a plane flight). This is not handled by Mobile IP. After all, this technology assumes the existence of a network, albeit a mobile one. System isolation assumes there is no network, mobile or static, so all procedures meant to circumvent the obstacles of a lossy, noisy medium are in vain: there simply is no medium. The system cannot use any form of IP.

System isolation is also different from network partitioning. Here, a collection of hosts may be able to reach some but not all destinations in the network. Since the network is still available (albeit in a diminished capacity), this condition is best handled by level 2 routers repairing the partition [PERLMAN 92]. A host in the partitioned region may receive ICMP error messages [POSTEL 81] indicating that the destination or the net is unreachable. During this time, it makes sense for this host to continue retrying the network access. Notice that this may also be the case if some of the interfaces of a multi-homed host are disabled (perhaps as a result of `ifconfig down`).

Finally, mobile systems often disconnect from the network and reconnect at another point using a vastly different medium (e.g. moving from a high-speed Ethernet LAN to a WAN using PPP). Furthermore, in the wireless case the network characteristics fluctuate dramatically even without changing media. These scenarios imply that a system may become weakly connected (i.e., service may diminish to the point that it is no longer usable, although strictly speaking the network is still available). Even though this is not described by system isolation, our prototype includes a condition notification facility that could accommodate this scenario (see Section 6).

3.0 Our Solution

The IP fast-fail capability prevents the system from freezing or retrying network operations when it is isolated. Network client processes become nonpersistent: instead of waiting for a reply from their server, they receive a notification that they are isolated from the network, and they immediately fail the operation. There are no futile retries.

Alternatively, clients can react to system isolation by making use of local caches or recording network operations to be replayed upon reconnection.

Our prototype implements this capability within the following constraints:

1. Accessing the loopback interface must not trigger the fast-fail mechanism. Only outgoing network accesses trigger it.
2. Existing, unmodified, applications should benefit from the fast-fail mechanism.

Our prototype works very well with existing applications (see section 7). When a system becomes isolated, there is no need to change its configuration to prevent it from issuing network requests. For example, the path environment variable may still point at network resources. These path queries fail immediately by virtue of IP fast-fail detection incorporated into NFS. The automounter, in particular, may try several servers in order to mount a certain file system over the network. Usual behavior implies a hang of several minutes until, one-by-one, all the servers time out. With fast-fail, there is no delay.

We have tested applications built with system isolation in mind. Sun's ROAM nomadic mail tool and a prototype disconnected cache file system now have an unequivocal and immediate notification that the system is isolated. They respond by entering their own disconnected operation mode.

We have also provided a framework for applications to subscribe to notifications of network conditions. This is for isolation aware applications that wish to make better use of this information.

4.0 Enabling and Disabling System Isolation

The system becomes isolated either:

1. explicitly, by direct user input requesting system isolation, or
2. implicitly, when all (non-loopback) interfaces are down or unusable.

Our prototype includes both a command line and a graphical user interface tool that allows the user to explicitly change the state of system isolation, and to inquire about the isolation status.

Implicitly enabling system isolation provides a capability for automatic reconfiguration if a smart interface detects network disconnection and configures itself down. If all interfaces configure themselves down, then the system becomes isolated. Notice that new semantics take effect only when ALL non-loopback interfaces are configured down. This allows the user the ability to selectively mark some interfaces down without causing system isolation.

For the SPARCstation Voyager, we have implemented a connection monitoring daemon that takes advantage of a special hardware feature to determine whether or not the Ethernet cable is plugged in. When the cable is removed, the daemon issues an `ifconfig down` on the interface. Usually, this is the only (non-loopback) interface the system uses. Consequently, the system automatically enters the isolated state. At this point, the system become perfectly usable in a stand-alone mode.

When the ethernet cable is plugged back in, the daemon senses it, and issues the equivalent of an `ifconfig up` on this interface. Since the number of *up* interfaces is no longer 0, the system is not isolated, and normal behavior resumes. Notice that the same is true if the user issues an `ifconfig up` on any interface (e.g., a PPP interface).

On systems without this special hardware support, a similar facility can be built into the Ethernet driver itself. When it detects no carrier on the physical interface, it can cause the interface to be marked down, which in turn causes the system to enter the isolated state. When the driver later senses a carrier, the interface can be brought up, resuming normal network operations.

5.0 Reacting to System Isolation

Once the system is isolated, IP's output routines generate an ICMP error report whenever an IP packet is sent to any interface except the loopback interface. The error message is returned back up the appropriate UDP, TCP, or ICMP protocol stack to the application.

5.1 Using ICMP to Propagate the Errors

We chose to use ICMP instead of a different mechanism because typical TCP/IP code already handles a similar case upon reception of an ICMP port unreachable message, and therefore it could be implemented without significant changes to existing network code. The only remaining question was *which ICMP error should we use to flag this condition*?

The original ICMP specification defines three other codes for destination unreachable errors:

- fragmentation needed
- host unreachable
- net unreachable

Of these, the last two appear related to fast-fail during isolated operation. However they are quite commonly used, and we did not want to overload their meaning. In fact, further investigation revealed we could not use them. RFC-1122 ("Requirements for Internet hosts - communication layers") specifies why [BRADEN 89a]:

> "A Destination Unreachable message that is received with code 0 (Net), 1 (Host), or 5 (Bad Source Route) may result from a routing transient and MUST therefore be interpreted as only a hint, not proof, that the specified destination is unreachable [IP:11]. For example, it MUST NOT be used as proof of a dead gateway (see Section 3.3.1)."

In short, the host and net unreachable codes are hints and are usually treated as *soft errors*. According to RFC-1122, these are simply recorded for eventual return if the connection times out. *Hard errors* abort the connection.

Even though [BRADEN 89a] defines additional codes:

6: destination network unknown
7: destination host unknown
8: source host isolated
9: communication with destination network administratively prohibited
10: communication with destination host administratively prohibited
11: network unreachable for type of service
12: host unreachable for type of service

it is not documented whether each of these should be treated as hard or soft [BRADEN 89b]. It seems like the code we could adopt for isolated operation is 8. This code was created for routers (actually, IMP's) for return to hosts [POSTEL 94].

As originally envisioned, reception of a source host isolated error from a router or IMP should be a hint. If one router informs the system that it is isolated, it does not mean that other paths (traversing other routers) are not available. However, if the source host isolated packet originates from the local system, as can be ascertained by checking the source address field, then we know the outgoing IP packet never made it into the network. This is no longer a hint, but a hard fact that the system does not have access to ANY network. In this case, we chose to treat this indication as a hard error.

There are two reasons why this use of the source host isolated error code does not introduce confusion.

1. Currently, this code is used very rarely, if at all. Not only is it considered obsolete [STEVENS 94], but current internet routers are required not to generate it [ALMQUIST 94].
2. Even if this code were received by the networking modules, it would not be interpreted as an indication of system isolation unless the source address of the ICMP packet corresponds to the local system. This limits the sender of the packet to the local system's fast-fail code.

In effect, the source address of the source host isolated packet determines the semantics to be adopted. For example, if the source is not local, 4.4BSD simply treats this message as equivalent to host unreachable. Thus, the user process displays the error message "No route to host". On the other hand, if the source is local, corresponding to the system isolation interpretation, the user sees the error message "Network is down".

5.2 Modifications to Network Modules

If there is an outgoing packet and the system is isolated, the IP module starts the chain of events by creating and returning upstream an ICMP unreachable error with code *source host isolated*.

The ICMP module was augmented to pass ICMP error notifications to the correct UDP stream. UDP handles the notification by producing a T_UDERROR_IND TLI message for consumption by its clients. Both UDP and TCP translate the ICMP error notification to the UNIX error ENETDOWN (127). Depending upon the state of a user selectable option, TCP may:

- record the ENETDOWN error to be returned when TCP finally times out, or
- kill the connection.

To do so, it uses the same function it calls to terminate timed out connections, but with the error ENETDOWN.

The sockets kernel module handles the T_UDERROR_IND message by sending an M_ERROR message up to the stream head. However, it only does so for connected sockets (i.e. those for which the source and destination information allow some sanity checks).

The next access to the socket returns with the appropriate error code, and the stream becomes unusable from this point on. The transport independent kernel module handles the T_UDERROR_IND by simply propagating it upstream.

Additionally, we made minor changes to `in.rdisc` (router discovery), `ping` and `snoop` for correct decoding of the source host isolated code. Router discovery was also made isolation aware by allowing it to bypass time-out loops if a source host isolated error is received.

6.0 A Proactive Approach

ICMP error messages certainly offer the advantage that the system already has the facilities to handle them. However, they are limiting in two important ways:

1. the type of information that can be carried
2. they are only sent in response to previous traffic.

In order to alleviate this, we have defined a set of ioctl commands for IP that allow isolation aware applications to query, set or reset the isolation status of the system. For example, NIS and DNS use the query mechanism to check the system's status before initiating network operations.

Applications may wish to be notified whenever the isolation status changes. Accordingly, a subscription mechanism was introduced for applications to receive immediate notification whenever IP's connectivity changes. We use this facility to implement a graphical application that displays real-time isolation status.

Another ioctl command with a similar purpose allows kernel modules to register a callback function with IP. When IP senses a change in isolation status, it invokes the registered function. A user-selectable option employing this facility allows the TCP module to abort all connections in the process of transmitting data.

These subscription and registration interfaces provide a framework for lower layers to notify upper layer protocols and applications of varying network conditions. For example, a cellular handoff notification could be used by TCP to "kick" its fast retransmit code [CACERES 94]. A good candidate for notification is the amount of bandwidth available. Each layer could use this information to decide if it is worth offering service. For example, if the bandwidth available is 2400 bps, TCP and UDP might still offer service, but NFS could just return an error. In effect, this allows the specification of minimum conditions for each type of network connection.

By generalizing these mechanisms we plan to implement a condition notification facility that separates *information* (e.g. "The bandwidth is now 4800 bps"), from *policy* (e.g. "What am I going to do about it?"). This gives each layer the freedom to make its own decisions. Of course, some layers depend on others. If in the above example UDP says: "This is too slow for me, I will send an error message upstream", NFS does not have much choice but to propagate an error upstream.

This enhanced condition notification facility will not only implement the current semantics for notification of system isolation and connectivity, but it will also accommodate additional conditions such as *moved* (to report mobile handoffs), *bandwidth* (for periodic updates of actual network bandwidth), *cost* (for connectivity pricing information), etc.

7.0 Application Response to Fast-Fail

One of the main motivations in our approach has been to allow unmodified applications to fast-fail. There may be two different responses depending on whether the application establishes its network connection before or after the system becomes isolated. We have tested both cases when applicable.

7.1 System Isolated before Issuing the Command

Usually, the application displays one of these two error messages:

1. Network is down
2. Unknown host

Message 1 above occurs when the application itself attempts the network operation. Usually, however, the application must first resolve a name. Depending on the system's configuration, DNS or NIS may contact a server on the application's behalf. They receive the error notification, and fast-fail in their efforts to resolve a name. The application itself also fast-fails, but this time it displays message 2 above. The results of our testing were:

- `rsh`, `rlogin`, `rcp`, `finger`

  Immediate exit after displaying one of the error messages above.

- `ftp`, `telnet`

  Immediate return to application prompt (`ftp>` or telnet>) after displaying one of the error messages above.

- spray

  Exits with an immediate failure notification to the user:

  "spray: cannot clnt_create <rhost>: netpath: RPC: Rpcbind failure - RPC: Unable to send"

  If resolving a name, the error message is:

  "spray: cannot clnt_create <rhost>: netpath: RPC: Unknown host"

- ping

  If resolving a name, immediate exit with message 2. Otherwise, it loops retrying to reach the target host, and displaying the ICMP host isolated error received from IP:

  "ICMP Source Host Isolated from gateway <localhost>..."

- rusers

  Immediate exit and message 2, if resolving a name. Otherwise, the following message:

  "RPC: Rpcbind failure"

- WWW browsers Mosaic and netscape

  Immediate notification formatted in html:

  "<H1>Fatal Error 500</H1>"

  The error may be caused by the application itself:

  "<B>Reason:</B> System call `connect' failed: Network is down."

  Or by name resolution via DNS or NIS:

  "<B>Reason:</B> Can't locate remote host: *host*."

- cd using the automounter

  This is a method of automatically mounting a file system when issuing a cd into it. It fast-fails with the following indication:

  "No such file or directory"

- cd into an NFS mounted file system

  The file system has already been mounted over the network (using the automounter or explicitly via mount). cd to the root of the mount point succeeds, because it requires no network traffic. All that is needed is a valid file handle. However, cd past the root of the mount point does require network traffic, and it fast-fails with these error messages.

  "NFS getattr failed for server <remote_host>: RPC: Program unavailable"

  "<target_file_system>: Network is down"

- cat NFS mounted file

  Immediate exit displaying:

  "cat: cannot open *file*"

- cd using OPENLOOK's filemgr into an NFS mounted file system

  Status display shows:

  "Unable to read directory for folder '*folder*': Network is down"

- edit an NFS mounted file with OPENLOOK's textedit

  Attempting to open an NFS mounted file brings up an error window:

  "The file `foo' does not exist. Please confirm creation of new file for textedit."

  Creating the new file exits the program with the following message:

  "NFS getattr failed for server <server>"

  "Unable to Save Current File. Cannot back-up file."

- ROAM nomadic mailtool

  Immediate notification in the status window:

  "Could not contact mail server"

  ROAM does not even display the login window. When instructed to connect, ROAM suggests the user try another mail server, as it has failed in trying to contact the default one. ROAM's disconnected mode is still available, though, and the user can compose messages, and queue them in the outbox.

7.2 System Isolated after Issuing the Command

Unless otherwise specified, the programs below exit immediately after the TCP connection is aborted. The usual behavior of exiting after a time-out is also available as a user-selectable option. The tests below were run with TCP aborts enabled.

- rsh, rlogin

  "Read error from network: Network is down"

  "Connection closed."

- `telnet`

  `"Connection closed by foreign host"`

- `ftp`

  `ftp` does not exit. Instead, it resets the connection, displays an error message:

  `"421 Service not available, remote server has closed connection"`

  and returns to the `ftp>` prompt.

- `rcp`

  `"rcp: lost connection"`

- `finger`

  The program displays only the information that made it before the network was severed. Since there is no error indication, this may be misleading.

- `spray`

  `"SPRAYPROC_GET: RPC: Unable to send; System error"`

- `rusers`

  `"RPC: Unable to send"`

- WWW browsers `Mosaic` and `netscape`

  If the http connection is severed during a transfer, there is no error indication. `netscape` even prints a completion status. The partial document is misleadingly displayed without any warnings. If the system becomes isolated after the http request has been sent but before the browsers receive the entire response from the server, there is no indication of failure. The browser waits until interrupted by the user.

- `cd` using OPENLOOK `filemgr` into an NFS mounted file system

  The status indicates that the

  `"Network is down"`

  Furthermore, the target directory's contents are displayed, but there is visual indication that not all the information was fetched. For example, if the file types are unresolved, their icons flag this condition.

- edit an NFS mounted file with OPENLOOK's `textedit`

  This can display a large number of error messages, specially when loading a file or scrolling. textedit can sometimes dump core and terminate. Simple operations like saving or reading simply generate a pop up error window:

- ROAM nomadic mailtool

  Immediate notification in the status window:

  `"Connection to server broken"`

  ROAM then enters its disconnected mode.

8.0 Programming Interface

This prototype provides isolation at the IP layer. An application can access IP either through a socket descriptor or a TLI file descriptor. In either case the application can receive the system isolation error through the standard error reporting facility supported by the transport layer.

In general, when an application sends or transmits a packet to the network, the library or system call completes successfully at the stream head, and then the data is forwarded down the stream. Thus, the application's transmit operation and the kernel's error reporting operation are asynchronous events. The next time the application accesses the socket or TLI file descriptor, the isolated condition is noticed and the system or library call exits with an appropriate error.

In most cases however, an application receives a system isolation indication prior to attempting a data transfer, because it typically uses NIS or DNS for name resolution. These detect the isolation and return an appropriate error to the calling program.

New applications can take advantage of the query ioctl discussed previously to ascertain whether or not the system is connected prior to initiating network activity.

Of course, if the application uses ICMP directly (ping, router discovery, etc.), it receives the ICMP packet generated to flag the isolation condition:

ICMP Type:	Destination Unreachable
Code:	8 (Source Host Isolated)

Applications that use UDP or TCP encounter the interfaces described below.

8.1 TLI

In the examples below, notice that the error is first noticed by some function setting `t_errno` to TLOOK, as is expected for all such asynchronous events [STEVENS 90].

Nevertheless, we could return TSYSERR and set errno to ENETDOWN. However, TLI already defines the TLOOK (event requires attention) mechanism to handle asynchronous events. Not doing so would depart from the interface specifications.

8.1.1 TCP case

System in isolated state before TCP connection is established.

The application issues `t_connect` to obtain a connection:

`t_connect`:

return value:	-1
t_errno:	TLOOK
errno:	0 (no error)

A subsequent call to `t_look` interrogates and clears the error condition as follows:

`t_look`:

return value:	T_DISCONNECT

TCP connection established before the system enters isolated state.

The application is already connected, so it sends data via `t_snd`, and succeeds. When TCP receives the subsequent ICMP host isolated error notification, it kills the connection. Upon reading the TLI endpoint via `t_rcv`, the error is noticed and the function call exits as follows:

`t_rcv`:

return value:	-1
t_errno:	TLOOK
errno:	0 (no error)

A subsequent call to `t_look` interrogates and clears the error condition as follows:

`t_look`:

return value:	T_DISCONNECT

If instead of reading or receiving, the application attempts to transmit again (`t_snd`) no error is reported. Errors are noticed when receiving

8.1.2 UDP case

TLI's connectionless send function is `t_sndudata`. If the system is isolated, this causes the ICMP error notification to propagate back from IP through UDP and the Transport Interface cooperating STREAMS module, `timod`(7). When the TLI file descriptor is accessed with a connectionless receive (`t_rcvudata`) the error is noticed:

`t_rcvudata`:

return value:	-1
t_errno:	TLOOK
errno:	0 (no error)

`t_look`:

return value:	0
t_errno:	T_UDERR (datagram error indication)

`t_rcvuderr`:

return value:	0
unit data error:	ENETDOWN (Network is down)

If instead of reading or receiving, the application attempts to transmit again (`t_sndudata`) no error is reported. Errors are noticed when receiving.

The TLI endpoint remains usable. This is different from the current sockets interface.

8.2 Sockets

As explained previously, the network isolation error is encountered after the send operation has completed successfully. The sockets interface returns such asynchronous errors in subsequent operations on the socket. Alternatively, the SO_ERROR option of `getsockopt` can be used to interrogate the error [STEVENS 90]. Datagram sockets must be connected in order to receive such error reports. Furthermore, these are only available via `getsockopt`, but not as a result of socket I/O calls (`recv`, `recvfrom`, `send`, `write`, `read`, etc). In practice, this SO_ERROR interface is not very useful because applications rarely call `getsockopt` before accessing the socket endpoint.

Instead, in the UDP case, we decided that accesses to the socket endpoint return with the error condition. We do so by:

1. setting so_error to the error reported in the T_UDERROR_IND message (ENETDOWN), and
2. sending an M_ERROR message (with

ENETDOWN being the error on both read and write operations) up to the stream head.

The issue here is that once an M_ERROR is seen at the stream head, that stream is rendered unusable (i.e. the sockets interface is destructive). This is the main difference with the TLI interface. The rules for pipes, FIFO's and sockets in this situation are very clear: if a write operation is attempted on an unwriteable descriptor, SIGPIPE is generated [STEVENS 90]. It is the application's responsibility to catch this signal (see also the man pages for `socket`(3N) and `write`(2)).

It is also possible to avoid sending an M_ERROR to the stream head in response to a host isolated indication. Setting the sockets error variable so_error to the error ENETDOWN does allow the applications to fetch this information by using the SO_ERROR option to getsockopt. However, this does not allow current unmodified applications to fast-fail on network operations.

Furthermore, there have been discussions about allowing the user to enable and disable isolation semantics on a per-socket basis. This would allow finer granularity in deciding which applications fast-fail. It also guarantees that unsuspecting applications do not malfunction in the presence of fast-fail. The interface to this socket option would be something like:

- Add a new setsockopt option at the SOL_SOCKET layer named SO_FF_ENABLED.
- If SO_FF_ENABLED option is set ON, the transport layer will disconnect upon receiving host isolated errors. TCP will abort the connection, and UDP will permanently error out the socket.
- If the SO_FF_ENABLED option is set to OFF, the transport layer behaves exactly as it does today (no disconnection semantics).

The default behavior would be to not fast-fail, implying that current applications would not behave any differently unless explicitly modified. Because of this, we decided not to implement this interface in our prototype.

The interface seen by sockets programmers is as follows:

8.2.1 TCP case

System in isolated state before TCP connection is established.

The application issues `connect` to obtain a connection:

`connect`:

return value:	-1
errno:	ENETDOWN (Network is down)

TCP connection established before the system enters isolated state.

The application is already connected, so it sends data via `send`, `sendto`, `sendmsg` or `write`, and succeeds. When TCP receives the subsequent ICMP host isolated error notification, it kills the connection. Upon reading the socket endpoint via `read`, the error is noticed and the function call exits as follows:

`read`, `recv`, `recvfrom` or `recvmsg`:

return value:	-1
errno:	ENETDOWN (Network is down)

If instead of reading or receiving, the application attempts another `write` operation (or `send`, `sendto`, `sendmsg`), the result is:

(another) `write`, `send`, `sendto` or `sendmsg`:

return value:	-1
errno:	EPIPE (Broken pipe)
signal generated:	SIGPIPE

By default, SIGPIPE terminates the calling program.

8.2.2 UDP case

There is no information to match unconnected UDP sockets with incoming error notifications. Accordingly, it is not desirable to fast-fail applications that use unconnected UDP sockets.

Connecting a datagram socket involves nothing more than local caching of some information. Since there is no exchange with the target system, this step always succeeds. Applications transmit data on a datagram socket via any of the `write`, `writev`, `send`, `sendto`, or `sendmsg` functions. Even if the system is isolated these will succeed. However, this causes the ICMP error notification to propagate back from IP through UDP and the sockets kernel module. The error is seen when the socket is accessed via any of `recv`, `recvfrom`, `recvmsg` or `read`.

`read`, `recv`, `recvfrom` or `recvmsg`:

return value:	-1
errno:	ENETDOWN (Network is down)

If instead of reading or receiving, the application attempts another `write` operation (or `send`, `sendto`, `sendmsg`), the result is:

(another) `write`, `send`, `sendto` or `sendmsg`:

return value:	-1
errno:	EPIPE (Broken pipe)
signal generated:	SIGPIPE

By default, SIGPIPE terminates the calling program. The DNS resolver library caused this problem, and would terminate the calling program. To avoid this we modified the resolver as outlined in section 9.

The SO_ERROR option to `getsockopt` behaves as follows:

`getsockopt`(SO_ERROR):

return value:	-1
errno:	ENETDOWN (Network is down)

Again, the function itself fails (returns "-1"), because the M_ERROR message is destructive to the stream head.

We experimented with another variation of UDP socket isolation that avoided sending an M_ERROR to the stream head. In this case, we obtained the following interface:

"non-destructive" `getsockopt`(SO_ERROR):

return value:	0
so_error:	ENETDOWN (Network is down)
errno:	not applicable

This is the expected behavior of `getsockopt`: the value of so_error is returned successfully. Of course, no other function calls (i.e. `write`, `read`, `send`, `recv`) detect the isolation indication, and in order to allow current unmodified applications to fast-fail, we opted for the M_ERROR message.

9.0 Modifications to Network Services and Utilities

In addition to making changes to allow the error to propagate back up the protocol stack we had to make changes to the network services modules and some utilities so that they would react properly (i.e. fast-fail) to system isolation.

As mentioned above, sockets may fast-fail by sending an M_ERROR message up the stream head. This had the unpleasant effect of causing DNS queries to receive SIGPIPE with the default effect of terminating the calling program without any error message. We modified DNS to check for the isolation status of the system before attempting network operations. If the system is isolated, an error is returned instead of attempting the network access. This avoids the SIGPIPE problem. Currently, the check is done by using the newly defined isolation status query ioctl for IP. A better way might be to subscribe via the status subscription ioctl. DNS would then know the exact status without having to constantly poll for it. A more general solution would be to allow M_ERROR messages to be interpreted as non-persistent conditions. The next operation would both fetch and clear the error condition. The socket would not be destroyed, and it would remain usable.

RPC exists as both a kernel module for use by NFS and as part of a library. We modified both instances of RPC to handle correctly the unit data error indication received from the underlying layer, and to propagate the error RPC_CANTSEND to its clients. Handling this error correctly at the kernel RPC module enables NFS to fast-fail. Handling it correctly at the RPC library enables RPC applications (e.g. NIS) to fast-fail.

There are two cases to take care of in NIS:

System in isolated state before the NIS domain has been bound.

> In this case, `ypbind` forks a child to broadcast for an NIS server. We decided not to allow the child to be spawned if the system was isolated. Accordingly, we query the network status in `ypbind` using one of the ioctl commands for IP. If the host is found isolated, a return of YPBIND_ERR_ERR causes NIS to fast-fail.

System in isolated state after the NIS domain has been *bound.*

We modified the ypbind program to recognize the RPC error RPC_CANTSEND, and return the NIS error YPBIND_ERR_ERR.

The `mount` program receives RPC_CANTSEND as a result of isolated operation, and quickly terminates without retrying.

The `talk` program did not fast-fail when initiating a new session while isolated, because it used a disconnected UDP socket. Since fast-fail only applies to connected UDP sockets, `talk` never received the ENETDOWN error indication. This was fixed by connecting the control socket used by `talk`.

10.0 Implementation Details

Our prototype primarily consists of modifications to Solaris 2.4, Sun's SVR4-based operating system. Since the TCP/IP implementation is STREAMS-based, we had to modify various networking modules as follows (with the number of additional lines of code in parentheses):

- IP kernel module, `ip` (724)
- UDP kernel module, `udp` (169)
- TCP kernel module, `tcp` (163)
- sockets kernel module, `sockmod` (199)
- Transport Interface cooperating STREAMS module, `timod` (9)
- RPC kernel module, `rpcmod` (15)

Furthermore, we modified the following user-level code:

- `ping` program (8)
- `snoop` program (22)
- `talk` program (3)
- router discovery demon, `in.rdisc` (63)
- DNS client library, `libresolv.so` (120)
- Network Services Library, `libnsl.so` (57)
- `ypbind` program (227)
- NFS `mount` program (53)

Administrative programs and scripts represent an additional 695 lines of code.

11.0 Conclusion

We have developed a prototype that allows systems usually connected to the network to continue to function in a diminished capacity if the network is disconnected. The prototype uses an ICMP code to provide a fast-fail capability that allows unmodified applications to fail quickly when a network connection is not available.

Our framework allows for isolation aware applications and kernel modules to proactively handle system isolation by registering for and receiving notification as soon as system conditions change.

Acknowledgements

We thank the third member of the team, Becky Wong, for her help throughout the project. In particular, she modified the NIS, NFS and the `mount` clients to fast-fail upon receiving the notifications sent by IP. Erik Nordmark and Bob Gilligan provided many valuable comments and suggestions.

References

[WALDO 94] Jim Waldo, Geoff Wyant, Ann Wollrath and Sam Kendall; A Note on Distributed Computing, Sun Microsystems Laboratories Technical Report #TR-94-29, November 1994.

[PERKINS 95] Charlie Perkins, ed., IP Mobility Support, Internet Draft, January 1995.

[POSTEL 81] Jon Postel, RFC-792, *Internet Control Message Protocol (ICMP)*, September 1981.

[PERLMAN 92] Radia Perlman, *Interconnections: Bridges and Routers*, Addison-Wesley, 1992.

[BRADEN 89a] Robert Braden, RFC-1122, *Requirements for Internet hosts-communication layers*, October 1989.

[BRADEN 89b] Robert Braden, RFC-1127, *Perspective on the Host Requirements RFCs*, October 1989.

[ALMQUIST 94] Philip Almquist and Frank Kastenholz, RFC-1716, *Towards Requirements for IP Routers*, November 1994.

[STEVENS 90] W.Richard Stevens, *UNIX Network Programming*, Prentice-Hall, 1990.

[STEVENS 94] W.Richard Stevens, *TCP/IP Illustrated: the protocols*, Addison-Wesley, 1994

[POSTEL 94] Jon Postel, private communication, Message-Id: <199411120056.AA21711@zephyr.isi.edu>, November 11, 1994.

[CACERES 94] Ramon Caceres and Liviu Iftode. *The Effects of Mobility on Reliable Transport Protocols*. In Proceedings of the 14th International Conference on Distributed Computing Systems, June 1994.